CSC148 - OO Design with Composition and Inheritance: Robot Strategy

Introduction

In this exercise, you will practice an OO design task that involves both composition inheritance. You will implement a Robot class that can make moves (`ATTACK`, `DEFENSE`, or `HOLD`) according to different strategies (normal, aggressive, or defense). Each move is generated randomly according to a probability distribution defined by the strategy. For example, a normal strategy generates ATTACK with probability 33%, DEFENSE with probability 33%, and HOLD with probability 34%; an aggressive strategy generates ATTACK with 100% probability.

Your goals is to come up with the proper OO design of the classes and complete the implementation in `robot_strategy.py`.

What to do

1. Read the starter code below (also available on the course website). Read the documentations and the TODO's carefully.
2. Discuss the following design questions with your group:
   - What classes need to be added to the design?
   - Which classes have the composition relation between each other?
   - Which classes have the inheritance relation between each other?
   - In the inheritance relation, what should be implemented in the base class and what should be implemented in the subclass? Why do you think your design is optimal?
   - What methods need to be implemented in the strategy classes? To answer this, read carefully how the strategy classes are used in the Robot class
3. After determining the proper design, complete the implementation together with your group.

```python
class Strategy:
    """
    A Strategy class

    === Attributes ===
    name: the name of the strategy
    attack_probability: the probability (%) of attacking, int between 0 and 100
    defense_probability: the probability (%) of attacking, int between 0 and 100
    (the probability of hold is calculated as (100 - attack - defense))

    === Representation Invariants ===
    - 0 <= self.attack_probability + self.defense_probability <= 100
    """
    # Attribute types
    name: str
    attack_probability: int
    defense_probability: int

    def __init__(self, name: str) -> None:

        self.name = name
        self.attack_probability = 0
        self.defense_probability = 0

    # TODO: complete the Strategy class. Think about the following:
    # - What methods should this class have? Look carefully at how this class is
    #   used by the Robot class
    # - If this class will be a base class, which methods' implementation should
    #   be in the base class and which ones should be the subclasses?


# TODO: add all the necessary subclasses of Strategy. Think about
# - What class are needed? Read carefull how these classes are used in the Robot
```

```
#   class.
# - To make your design optimal, any implementation that's shared by the
#   subclasses should be written only once in the base class and inherited. Each
#   subclass should only do what's special about itself.


class Robot:
    """
    A Robot class

    === Attributes ===
    name: the name of the robot
    strategy: the current strategy that is adopted by the robot
    """
    # Attribute types
    name: str
    # TODO: what other attributes does a robot have?

    def __init__(self, name: str) -> None:
        """
        Create a robot with a name and a default strategy of NormalStrategy
        """
        self.name = name
        # TODO: initialize any other attributes

    def set_strategy(self, strategy: Strategy) -> None:
        """
        Set the current strategy to <strategy>
        """
        print("Setting strategy to {} ...".format(strategy.get_name()))
        # TODO: complete this

    def move(self) -> None:
        """
        Make a move by printing a string
        """
        next_move = "wrong move" # TODO: fix this line
        print("{} made a move to {}!".format(self.name, next_move))


if __name__ == "__main__":

    robot = Robot("Cocomelon")

    # The following 10 moves should be performed with the normal strategy
    # A normal strategy means 33% attack, 33% defense, and 34% hold
    for _ in range(10):
        robot.move()

    robot.set_strategy(AggressiveStrategy())
    # The following 10 moves should be performed with the aggressive strategy
    # An aggressive strategy means 100% attack
    for _ in range(10):
        robot.move()

    robot.set_strategy(DefensiveStrategy())
    # The following 10 moves should be performed with the aggressive strategy
    # A defensive strategy means 100% defense
    for _ in range(10):
        robot.move()
```

```
# Below is an example printout
"""
Cocomelon made a move to DEFENSE!
Cocomelon made a move to HOLD!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to HOLD!
Cocomelon made a move to HOLD!
Cocomelon made a move to ATTACK!
Cocomelon made a move to DEFENSE!
Setting strategy to aggressive ...
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Cocomelon made a move to ATTACK!
Setting strategy to defensive ...
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
Cocomelon made a move to DEFENSE!
"""
```